[redacted]

UIUCDCS-R-76-817 Math COO-2383-0034 8

Il6N
no. 817
Cop 2

ALGORITHMS FOR SPARSE GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING

by

Andrew H. Sherman

July 1976

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-76-817 COO-2383-0034

ALGORITHMS FOR SPARSE GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING

by

Andrew H. Sherman

July 1976

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMAPIGN
URBANA, ILLINOIS 61801

Supported in part by the National Science Foundation under grant NSF DCR-73-07998 and by the Energy Research and Development Administration under grant US ERDA E(11-1) 2383. An earlier version of this paper was presented at the 1976 SIAM National Meeting.

Abstract

In this paper we compare several algorithms for sparse Gaussian elimination with column interchanges. The algorithms are all derived from the same basic elimination scheme, and they differ mainly in implementation details. We examine their theoretical behavior and compare their performances on a number of test problems with that of a high quality complete threshold pivoting code. Our conclusion is that partial pivoting codes perform quite well and that they should be considered for sparse problems whenever pivoting for numerical stability is required.

Key Words: Sparse Gaussian Elimination, Sparse Linear Systems, Linear Equations, Analysis of Algorithms, Pivoting Algorithms

CR Categories: 4.6, 5.14, 5.25

[illegible]

[illegible]

[illegible]

1. Introduction

Let A be an NxN nonsingular matrix, and consider the system of linear equations

$$A\ x = b \qquad (1.1)$$

where x and b are vectors of length N. If N is small or A is dense (i.e. most entries a_{ij} are nonzero), then the standard algorithm for solving (1.1) is Gaussian elimination with partial pivoting (cf. Forsythe and Moler [6]). This algorithm is quite well understood, from both the theoretical and practical points of view, and it produces an accurate solution x (provided that A is well conditioned and the equations are well scaled, both of which we shall assume throughout).

When N is large and the matrix A is sparse (i.e. most entries a_{ij} are zero), the situation is quite different. Now there are two problems to contend with: the accurate computation of x, and the exploitation of the large number of zeroes in A to reduce storage and computing time. When A is symmetric and positive definite, a number of methods may be effective, including variants of Gaussian elimination and iterative methods such as conjugate gradient. However, in most other cases, Gaussian elimination with pivoting seems to be the best method currently available.

In this paper we examine algorithms for sparse Gaussian elimination with partial pivoting. In Section 2 we introduce a basic high-level algorithm and point out several problems that arise in practice. In Section 3 we discuss several possible algorithm refinements which solve the practical problems, and we give a theoretical comparison of them based on their behavior in the best, average, and worst cases. Unfortunately, we do not believe that our theoretical analysis reflects what happens in practice, so Section 4 we give the results of some numerical experiments with our algorithms and compare them with one of the better existing codes for Gaussian elimination with pivoting. Finally, we conclude in Section 5 by suggesting directions for future development based on the results in this paper.

2. A Basic Algorithm

In this section we examine a basic algorithm for sparse Gaussian elimination with partial pivoting. More precisely, we consider an algorithm which performs Gaussian elimination with column interchanges on the matrix A to effectively obtain a factorization of the form

$$A\,Q = L\,U, \tag{2.1}$$

where L is lower triangular, U is unit upper triangular, and Q is a permutation matrix corresponding to the column interchanges. Once this factorization has been obtained, we can compute x by solving the systems

$$L\,y = b \text{ and } U\,Q^T\,x = y,$$

although we will not discuss this forward and back solution process here. The algorithm we give is a modification of an algorithm without pivoting discussed in [5].

```
line
 1    For $k = 1$ to $N$ do
 2      ($I_k = \{j:\ a_{kj} \neq 0\}$;
 3       $m = \min \{j:\ j \in I_k \cup \{N + 1\}\}$;
 4       While $k > m$ do
 5         ($I_k = I_k - \{m\} \cup I_m$;
 6          For $j \in I_m$ do
 7            ($a_{kj} = a_{kj} - a_{km} \cdot a_{mj}$);
 8          $m = \min \{j:\ j \in I_k \cup \{N + 1\}\}$);
 9       If $m \leq N$ then do
10         ($\ell = \min \{j:\ |a_{kj}| = \max \{|a_{ki}|:\ k \leq i \leq N\}\}$;
11          If $a_{k\ell} = 0$ then exit with failure (zero pivot);
12          Interchange columns $k$ and $\ell$;
13          If $m = k$ then $I_k = I_k - \{k\}$;
14          Else $I_k = I_k - \{\ell\}$;
15          For $j \in I_k$ do
16            ($a_{kj} = a_{kj}/a_{kk}$));
17       Else exit with failure (zero pivot));
```

Algorithm 2.1

The basic algorithm, Algorithm 2.1, is a row-oriented version of Gaussian elimination with column interchanges. It consists of N steps, during each of which one row of A is processed. When processing the k-th row at the k-th step, I_k is used to hold the indices of all columns containing nonzeroes in the k-th row. Initially (line 2), these are just the nonzero columns of A in the k-th row. Then, in increasing order, for each $m \in I_k$, $m < k$, a multiple of row m of U is subtracted from row k to annihilate a_{km} (lines 3 - 8). This may cause fill-in, i.e., the introduction of new nonzero elements in the k-th row, so I_k must be updated (line 5). Finally, when all $m < k$ have been processed, I_k contains the indices of columns which contain nonzeroes in the portion of the k-th row lying in the upper triangle of U. If A is nonsingular, I_k will not be empty, so that $m \leqslant N$ will hold in line 9. In that case the algorithm selects the remaining nonzero element with maximum modulus and interchanges its column with the k-th column. If A is singular, on the other hand, then either $m = N + 1$ in line 9, or $a_{k\ell} = 0$ in line 11. In either case, the algorithm takes a failure exit. (However, this situation could simply be handled by storing a null k-th row, if one wished to deal with singular matrices.)

That Algorithm 2.1 is numerically stable can be shown quite easily by relating it to the application of standard Gaussian elimination with row interchanges to A^T. In fact, assume that such a procedure produces a factorization of A^T as

$$P\,A^T = \tilde{L}\,\tilde{U}, \tag{2.2}$$

where $\tilde{L}$ is unit lower triangular, $\tilde{U}$ is upper triangular, and P is a permutation matrix corresponding to the row interchanges. Then we can show that Algorithm 2.1 produces the factorization (2.1) of A with $L = \tilde{U}^T$, $U = \tilde{L}^T$, and $Q = P^T$. Since the computation of (2.2) is numerically stable (cf. [6]), that of (2.1) by Algorithm 2.1 is also.

In Algorithm 2.1 there are only two operations whose costs can be greatly affected by the column interchanges due to pivoting: the selection of m in lines 3 and 8, and the updating of I_k in line 5. The costs of these operations may depend critically on the representation of the sets

I_k and I_m (e.g. ordered vs. unordered, list vs. array), while the costs of other operations in the algorithm in general do not.

As an illustration, consider two possible representations of I_k: as a linked list in increasing order and as a heap with smaller nodes nearer the root (cf. Knuth [7], p. 145). In the first case, selecting m is easy, since we want to choose the first entry of I_k, and the updating operation in line 5 can be performed as a linear merge of the two ordered lists in $O(|I_m| + |I_k|)$ time. (We use $|I_m|$ and $|I_k|$ to denote the number of entries in $|I_m|$ and $|I_k|$, respectively, when we start to merge I_m into I_k.) In the second case, selecting m is still easy, since we want to choose the root of the heap, but now the updating operation may be more costly, since we must delete m and separately insert each entry of I_m into the heap for I_k. This may require a total of $O(|I_m| \log_2(|I_m| + |I_k|))$ time.

From this example it might seem that there is no question as to the proper course: simply use an ordered linked list representation for I_k (and an ordered array representation for previously computed sets I_m). In fact, if there were no pivoting, this conclusion would probably be correct. However, as soon as column interchanges can occur, the situation is not so clear-cut. The problem is that while I_m was in increasing order when it was computed at the m-th step, subsequent pivoting operations may cause it to be out of order at the k-th step. Thus the set union operation can no longer be performed as a simple ordered list merge. We might try to avoid this problem by keeping the sets I_k in some canonical order (say the initial column order of A), but then the selection of m would become difficult, since it would require a (necessarily linear) search for the minimum remaining column index in I_k (in the current column order) at the k-th step.

In the remainder of this paper, we will use Algorithm 2.1 as the basis for our discussion of various alternative means of implementing the selection and updating operations which seem so problematical when pivoting is permitted. The next section contains descriptions of several possible implementations and a limited theoretical discussion of them. In Section 4, we will present more meaningful comparisons, based on numerical experiments.

3. Algorithm Implementations

In this section we present six different ways of implementing the selection and updating operations of Algorithm 2.1. For each we will briefly describe the representations used for I_k and I_m (i.e. for I_k during the k-th step and for the previously computed sets I_m). Then we will outline the methods used to select m and to form the union $I_k \cup I_m$, giving rough estimates of the costs of these operations in the best case (no pivoting), the average case, and the worst case. By average case here, we mean the average case with respect to the N! possible reorderings of the columns of A, not with respect to all possible matrices A. While the former may be less interesting, the latter seems all but impossible to study except through some (probably unrealistic) testing program using randomly generated matrices A.

1) Run Insertion

Each set I_k is kept as a linked list in increasing order with respect to the current column order at the k-th step, and the sets I_m are kept as arrays in increasing order with respect to the column order at the m-th step, where they were computed. The selection of m requires constant time, since m is always the first remaining entry of I_k. To form $I_k \cup I_m$, we effectively split I_m into its component increasing runs (cf. Knuth [7], p. 34) and separately merge each of the runs into I_k. In practice, this is done by merging the entries of I_m into I_k and backing up to the beginning of I_k whenever an out-of-order entry is found in I_m. In the best case, the union operation requires $O(|I_m| + |I_k|)$ time, and in both the average and worst cases it requires $O(|I_m| \cdot |I_k|)$ time (cf. Knuth [7], pp. 34-45, for a discussion of the average number of runs in a sequence).

2) <u>List Merge with Bubble Sort</u>

The sets I_k and I_m are kept as in (1), except that I_m is reordered whenever it is used. As before, choosing m requires constant time. To form $I_k \cup I_m$, we first sort I_m with a bubble sort (cf. Knuth [7], pp. 106-110) and then perform an ordered list merge. In the best case, this will require $O(|I_m| + |I_k|)$ time, while in the average and worst cases it will require $O(|I_m|^2 + |I_k|)$ time.

3) <u>List Merge with Insertion Sort</u>

This is identical with (2), except that I_m is sorted with a straight insertion sort (cf. Knuth [7], pp. 80-82) when it is used. As above, m can be chosen in constant time, and the set union operation requires $O(|I_m| + |I_k|)$ time in the best case and $O(|I_m|^2 + |I_k|)$ time in the average and worst cases.

4) <u>List Merge with Shell Sort</u>

This is also identical with (2), except that I_m is sorted with a Shell sort (cf. Knuth [7], pp. 84-95) using increments as suggested in

Knuth [7], p. 95. Again m can be chosen in constant time, and the set union operation requires $O(|I_m|^2 + |I_k|)$ time in the worst case. In the average case, however, only $O(|I_m|^{3/2} + |I_k|)$ time is required for $|I_m| \leq 121$.

5) <u>Heap Insertion</u>

The set I_k is split into two parts. $I1_k = \{m \in I_k: \; m \leq k\}$ and $I2_k = \{m \in I_k: \; m > k\}$. $I1_k$ is kept as a heap with smaller nodes nearer the root (cf. Knuth [7], p. 145), and $I2_k$ is kept as an unordered array. In addition the characteristic vector of I_k (cf. Aho, Hopcroft, and Ullman [1], p. 49) is kept to avoid storing duplicate entries. At the end of the k-th step, $I1_k$ is empty, and $I_k = I2_k$. Hence we assume that the sets I_m are stored as unordered arrays. (In fact, there is no loss in not having the sets I_m ordered.) Choosing m in this case still requires constant time, since we want to choose the root of the heap for $I1_k$. To form $I_k \cup I_m$, we separately add each entry of I_m either to the heap for $I1_k$ or to the end of the unordered array for $I2_k$. To insert an entry into the heap requires $O(\log_2 |I1_k|)$ time, and to add an entry to the unordered array requires constant time. Thus we expect that the best case time for the set union operation will be $O(|I_m|)$ time (when all the entries of I_m are larger than k), and that the average and worst case times will be $O(|I_m| \log_2 (|I_m| + |I_k|))$. Note that deleting m from I_k requires $O(\log_2 |I_k|)$ time, a usually insignificant cost.

6) <u>Pivot Search (cf. Curtis and Reid [3])</u>

The set I_k is kept as a linked list in increasing order with respect to a canonical ordering (e.g., the initial column order of A), and the sets

I_m are kept as ordered arrays in the canonical order. Now choosing m requires searching I_k for the minimum remaining column number in current order, and since this must be done as a linear search, it will require $O(|I_k|)$ time on average. In return for this searching, we can again form $I_k \cup I_m$ in $O(|I_m| + |I_k|)$ time independent of the amount of pivoting, since both I_m and I_k are in increasing order with respect to the canonical order.

4. Numerical Results

In this section we give numerical results obtained by programming the five implementations of Section 3 and running the programs on several test problems. The results here are not comprehensive, but the problems are realistic ones. The problem GS192 arose in the solution of a parabolic partial differential equation and was supplied by Professor Paul Saylor of the University of Illinois. The others were taken from a collection of test problems supplied by Dr. Iain Duff of the Atomic Energy Research Establishment at Harwell, England. Brief descriptions of the problems are given in Table 4.1. (See Duff and Reid [4] for further discussion of the Harwell test problems.)

Problem	N	Application	Remarks
ARC130	130	Laser Research	
SHL0	663	Linear Programming	Lower triangular
SHL200	663	Linear Programming	
SHL400	663	Linear Programming	
STR0	363	Linear Programming	Lower triangular
STR200	363	Linear Programming	
STR400	363	Linear Programming	
GS192	192	Parabolic Partial Differential Equation	Banded with half bandwidth of 19

TABLE 4.1

Experiments have shown that the storage and time costs of the algorithms discussed here may depend heavily on the initial row and column orders of A. Except for GS192, we chose to order the rows in order of increasing number of nonzero entries and to order the columns symmetrically. For GS192, we used the matrix exactly as supplied to us. If the permutation matrix P accomplishes the desired row reordering, then we applied the algorithms to the permuted system

$$P\ A\ P^T(P\ x) = P\ b.$$

There is little theoretical reason to expect that the choice of P would have a significant effect on the accuracy of the computed solution x, except that reducing the number of arithmetic operations performed should give a corresponding reduction in the roundoff error incurred. In practice, however, we found that particularly inefficient orderings of the rows led to poor accuracy, while orderings which were at least reasonably efficient gave uniformly good results. Unfortunately, at present we can say no more than this, but we hope to investigate further the problem of choosing the initial row and column orderings. (Note, however, that the simple row reorderings described above gave good numerical results for our test problems.)

To provide a benchmark comparison for our algorithms, we solved the test problems with a code due to A. R. Curtis and J. K. Reid [2] which is part of the Harwell Subroutine Library. This code uses a rather more sophisticated complete pivoting algorithm to factor A and compute x. Where our algorithms pre-order A to exploit sparseness and then pivot entirely for numerical stability, the Harwell code uses a strategy known as threshold pivoting to do both at once.

In threshold pivoting the pivot element at the k-th step is chosen to be that entry a_{ij}, $k \leq i, j \leq N$, which requires the fewest arithmetic operations to eliminate and which satisfies either

$$|a_{ij}| \geq u \cdot \max \{|a_{i\ell}|: \ k \leq \ell \leq N\} \text{ or}$$

$$|a_{ij}| \geq u \cdot \max \{|a_{\ell j}|: \ k \leq \ell \leq N\} ,$$

where u is a threshold parameter satisfying $0 < u \leq 1$. For our experiments we used two different settings for u: u = 1.0E-6, to allow pivoting almost entirely to exploit sparseness, and u = 1.0, to force pivoting almost entirely for numerical stability. Note that the form of the threshold test allows for some flexibility in either case. For intermediate values of u, we would expect the code to exhibit a behavior somewhere between the two extremes we used.

Our experiments were run in single precision on an IBM System/360 Model 75 using the Fortran IV Level G compiler. All of the programs gave equally accurate results as measured by the L_2 and L_∞ norms of both the absolute error in the solution and the residual, except that the Harwell code with u = 1.0E-6 was sometimes less accurate than the others. Hence the important question in our tests was efficiency, and we chose to use two measures: the number of nonzeroes in the computed LU factorization (as some measure of storage cost), and the time required to compute the solution x from the original (not yet pre-ordered) system (1.1). We should note that probably all of the codes could be made somewhat faster by more careful coding and that the times may be in error by approximately 10% due to system overhead. However, the results are still qualitatively interesting.

The results shown in Table 4.2 show several things. Most obvious, the partial pivoting codes are invariably much faster than the Harwell code, even though they produce a denser factorization (and hence perform more arithmetic operations). Next, it appears that the Shell Sort, Heap Insertion, and Pivot Search algorithms performed worse than the other partial pivoting algorithms on most of the problems. In the first case this is probably due to the nature of the data which must be sorted; in the second case, this is probably due to data structure overhead; and in the last case this is probably due to the fact that the costly searches occurred in the filled-in rows at each step. Finally, it seems that the four remaining partial pivoting algorithms all performed comparably well, with the Run Insertion algorithm appearing best over all, perhaps because it can be implemented with little programming overhead.

Method / Problem		Harwell u = 1.0	Harwell $u = 10^{-6}$	Run Insertion	Bubble Sort	Insertion Sort	Shell Sort	Heap Insertion	Pivot Search
ARC130	NZ	1293	1293	1457	1457	1457	1457	1457	1457
	Secs.	6.89	7.36	0.48	0.56	0.60	0.64	0.58	0.70
SHL0	NZ	1687	1687	2568	2568	2568	2568	2568	2568
	Secs.	5.90	7.09	1.38	0.99	1.13	1.10	4.59	2.11
SHL200	NZ	1726	1726	2637	2637	2637	2637	2637	2637
	Secs.	6.47	6.31	1.18	1.12	1.13	1.23	4.50	2.21
SHL400	NZ	1712	1712	2651	2651	2651	2651	2651	2651
	Secs.	6.36	7.09	1.19	1.08	1.04	1.17	4.34	2.16
STR0	NZ	2454	2454	2953	2953	2953	2953	2953	2953
	Secs.	3.83	4.49	0.61	0.48	0.50	0.48	1.90	0.73
STR200	NZ	3640	3516	7254	7254	7254	7254	7254	7254
	Secs.	8.91	8.22	2.04	3.32	3.48	4.61	3.48	4.61
STR400	NZ	3879	3735	8233	8233	8233	8233	8233	8133
	Secs.	10.91	9.58	2.43	3.45	3.82	5.14	3.89	4.35
GS192	NZ	6568	6746*	6880	6880	6880	6880	6880	6880
	Secs.	60.92	82.42*	10.22	14.41	19.97	35.79	10.57	17.95

* The method failed to compute an accurate solution.

TABLE 4.2

5. Conclusion

There seem to be several directions for future research in this area. On the theoretical side, there is a particularly troubling problem which may be stated as follows. Let C(N) be the cost (in time) of solving a sparse system (1.1) without pivoting. (Assume for a moment that this is numerically stable.) It would be desirable for a pivoting algorithm to have a cost which was $O(N \cdot C(N))$ in the worst case and which declined to C(N) as the amount of pivoting decreased. Unfortunately, none of the algorithms discussed here has both properties, and, in fact, it is unclear whether any algorithm could have them.

In a more practical vein, it is desirable to continue the testing and comparison program begin in the work reported in this paper. One problem of importance is the selection of the initial row ordering. Another question which might be investigated is whether there is some threshold technique which might be applied to partial pivoting as discussed here. Since no information is kept about the column structures of A, L, and U, the strategy used in the Harwell code is not suitable, but it might be possible to use a threshold parameter to decide whether or not to pivot at all in a certain row. Reducing the amount of pivoting in this way might make the algorithms run faster without significantly affecting accuracy.

Finally, we must turn to the eventual goal of work such as ours, the production of useful mathematical software. It is true that the behavior of a partial pivoting code depends quite heavily on the specific problem being solved, in particular on the amount of pivoting required and on the initial ordering of the rows and columns of A. Hence our sample of problems may provide a biased and incorrect view of the true situation. However, we believe that our results do provide enough encouragement to justify the development and extensive testing of high quality subroutines based on one or more of our algorithms.

6. References

[1] A. Aho, J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[2] A. R. Curtis and J. K. Reid. FORTRAN Subroutines for the Solution of Sparse Sets of Linear Equations. AERE Report R6844, Harwell, England, 1971.

[3] A. R. Curtis and J. K. Reid. The Solution of Large Sparse Unsymmetric Systems of Linear Equations. Information Processing 71, pp. 1240-1245, 1971.

[4] I. S. Duff and J. K. Reid. A Comparison of Sparsity Orderings for Obtaining a Pivotal Sequence in Gaussian Elimination. AERE Report T. P. 526, Harwell, England, 1973.

[5] S. C. Eisenstat and A. H. Sherman. Efficient Implementation of Sparse Nonsymmetric Gaussian Elimination Without Pivoting. SIGNUM Newsletter 10:26-29, 1975.

[6] G. E. Forsythe and C. B. Moler. Computer Solution of Linear Algebraic Systems. Prentice-Hall, 1967.

[7] D. E. Knuth. The Art of Computer Programming, Volume 3: Searching and Sorting. Addison-Wesley, 1973.

Form AEC-427
(6/68)
AECM 3201

U.S. ATOMIC ENERGY COMMISSION
UNIVERSITY-TYPE CONTRACTOR'S RECOMMENDATION FOR DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

(See Instructions on Reverse Side)

1. AEC REPORT NO.
COO-2383-0034

2. TITLE
ALGORITHMS FOR SPARSE GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING

3. TYPE OF DOCUMENT (Check one):

[✓] a. Scientific and technical report
[ ] b. Conference paper not to be published in a journal:
Title of conference
Date of conference
Exact location of conference
Sponsoring organization
[ ] c. Other (Specify)

4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

[✓] a. AEC's normal announcement and distribution procedures may be followed.
[ ] b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.
[ ] c. Make no announcement or distrubution.

5. REASON FOR RECOMMENDED RESTRICTIONS:

6. SUBMITTED BY: NAME AND POSITION (Please print or type)
C. W. Gear
Professor and Principal Investigator

Organization
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Signature [illegible]

Date July 1976

FOR AEC USE ONLY

7. AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION:

8. PATENT CLEARANCE:

[ ] a. AEC patent clearance has been granted by responsible AEC patent group.
[ ] b. Report has been sent to responsible AEC patent group for clearance.
[ ] c. Patent clearance not required.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-76-817	2.	3. Recipient's Accession No.
4. Title and Subtitle ALGORITHMS FOR SPARSE GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING			5. Report Date July 1976
			6.
7. Author(s) Andrew H. Sherman			8. Performing Organization Rept. No. UIUCDCS-R-76-817
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. US ERDA E(11-1) 2383
12. Sponsoring Organization Name and Address United States Energy Research and Development Administration Chicago Operations Office 9800 South Cass Avenue, Argonne, IL 60439			13. Type of Report & Period Covered
			14.

15. Supplementary Notes

16. Abstracts

In this paper we compare several algorithms for sparse Gaussian elimination with column interchanges. The algorithms are all derived from the same basic elimination scheme, and they differ mainly in implementation details. We examine their theoretical behavior and compare their performances on a number of test problems with that of a high quality complete threshold pivoting code. Our conclusion is that partial pivoting codes perform quite well and that they should be considered for sparse problems whenever pivoting for numerical stability is required.

17. Key Words and Document Analysis. 17a. Descriptors

Sparse Gaussian Elimination, Sparse Linear Systems, Linear Equations, Analysis of Algorithms, Pivoting Algorithms

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 21
	20. Security Class (This Page UNCLASSIFIED	22. Price